Referencing The Unicode Standard

Document #:	P2736R2
Date:	2023-02-09
Programming Language C++	
Audience:	SG-16

Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

Abstract

We propose to reference The Unicode Standard instead of ISO 10646. This proposal has no impact on implementations. This resolves NB comments FR-010-133 and FR-021-013

Revisions

R2

- Wording improvements following SG16 feedback.
- Provide editorial guidance to resolve the merge conflict with P2713R1 [1]. - Applied Feedback from Core.

R1

- Wording improvements following SG16 feedback.

R0

Initial revision

Motivation

Unicode is developed as The Unicode Standard by The Unicode Consortium. From that is derived the ISO 10646 standard. However, Unicode has a much larger scope as it describes algorithms for normalization, clusterization, comparison, case transformation, bi-directional text handling, and more. C++ makes use of these features.

As such, in the current state, we reference ISO 10646 and Unicode, and The Unicode annexes. Care must be taken that the versions referenced are coherent as Unicode algorithms are designed to work with a matching specification and accompanying Unicode data. This is challenging as Unicode and ISO 10646 have different release cycles.

Additionally, there are subtle differences in terminology, the main impact of which is to consume time in study groups. It is also an issue for implementers who usually reference

Unicode directly as Unicode delivers the necessary Unicode data in an easily extractible and toolable format (whereas 1SO 10646 is just a PDF).

By referring to a single Unicode Standard, we can reduce the number of references that we have to maintain (and thereby the number of dependencies the standard carries), as well as making the standard clearer.

Difference between Unicode and ISO 10646

The Unicode standard explains:

> The Unicode Consortium maintains a strong working relationship with ISO/IEC JTC1/SC2/WG2, the working group developing International Standard 10646. Today both organizations are firmly committed to maintaining the synchronization between The Unicode Standard and ISO/IEC 10646. Each standard nevertheless uses its own form of reference and, to some degree, separate terminology.

> ISO/IEC 10646:2011 has significantly revised its discussion of encoding forms, compared to earlier editions of that standard. The terminology for encoding forms (and encoding schemes) in 10646 now matches exactly the terminology used in The Unicode Standard. Furthermore, 10646 is now described in terms of a codespace U+0000..U+10FFFF, instead of a 31-bit codespace, as in earlier editions. This convergence in codespace description has eliminated any discrepancies in possible interpretation of the numeric values greater than 0x10FFFF.

As such, the definition of UTF-8, UTF-16, and UTF-32, as well as the set of supported abstract characters and their code point value is the same. Most of the terminology changes operated consist of replacing ”UCS” with ”Unicode”.

There are, however, a few points to mention.

ISO 10646 defines ”character” as

> member of a set of elements used for the organization, control, or representation of textual data Note 1 to entry – A graphic symbol can be represented by a sequence of one or several coded characters.

Unicode defines instead ”abstract character”

> A unit of information used for the organization, control, or representation of textual data.

This is a positive change as the term character is both defined by the C++ standard and constantly misused. in the long term, having a term to talk about an abstract character when we mean it is a positive change.

Conformance with Unicode

Unicode specifies a list of conformance requirements (Chapter 3.2)

- C1: *A process shall not interpret a high-surrogate code point or a low-surrogate code point as an abstract character.*
  C++ satisfies this requirement.
- C2: *A process shall not interpret a noncharacter code point as an abstract character*
- C3: *A process shall not interpret an unassigned code point as an abstract character.*

  We create characters out of thin air for specifying the translation set. However, we never exchange or materialize, or use these things except for specification purposes.
- C4: *A process shall interpret a coded character sequence according to the character semantics established by this standard if that process does interpret that coded character sequence.*
- C4: *A process shall not assume that it is required to interpret any particular coded character sequence.*
- C6: *A process shall not assume that the interpretations of two canonical-equivalent character sequences are distinct*
- C7: *When a process purports not to modify the interpretation of a valid coded character sequence, it shall make no change to that coded character sequence other than the possible replacement of character sequences by their canonical-equivalent sequences.*

  C++ satisfies these requirements.
- C8: *When a process interprets a code unit sequence which purports to be in a Unicode character encoding form, it shall interpret that code unit sequence according to the corresponding code point sequence.*
- C9: *When a process generates a code unit sequence which purports to be in a Unicode character encoding form, it shall not emit ill-formed code unit sequences.*

  C++ satisfies these requirements.
- C10: *When a process interprets a code unit sequence which purports to be in a Unicode character encoding form, it shall treat ill-formed code unit sequences as an error condition and shall not interpret such sequences as characters*

  C++ satisfies this requirement, arguably. The C functions do behave weirdly in the presence of invalid code units, but do they actually purports to support Unicode if they don't support multi bytes encodings?
- C11: *When a process interprets a byte sequence which purports to be in a Unicode character encoding scheme, it shall interpret that byte sequence according to the byte order and specifications for the use of the byte order mark established by this standard for that character encoding scheme*

  I think we do.

- C14: *A process that tests Unicode text to determine whether it is in a Normalization Form shall do so in accordance with the specifications in Section 3.11, Normalization Forms*
- C16: *Normative references to The Unicode Standard itself, to property aliases, to property value aliases, or to Unicode algorithms shall follow the formats specified in Section 3.1, Versions of The Unicode Standard.*

  This requirement should be satisfied with the adoption of this paper. Care must be taken when referencing properties.
- C17: *Higher-level protocols shall not make normative references to provisional properties.* • Higher-level protocols may make normative references to informative properties.
- C17: *If a process purports to implement a Unicode algorithm, it shall conform to the specification of that algorithm in the standard, including any tailoring by a higher-level protocol as permitted by the specification.*

  Yes.

Omitted requirements are not applicable.

Terminology

- D4 Character name: A unique string used to identify each abstract character encoded in the standard.
- D5 Character name alias: An additional unique string identifier, other than the character name, associated with an encoded character in the standard.
- D7 Abstract character: A unit of information used for the organization, control, or representation of textual data.
- D10 Code point: Any value in the Unicode codespace.
- D11 Encoded character: An association (or mapping) between an abstract character and a code point.
- D76 Unicode scalar value: Any Unicode code point except high-surrogate and low-surrogate code points.
- D77 Code unit: The minimal bit combination that can represent a unit of encoded text for processing or interchange.
- D78 Code unit sequence: An ordered sequence of one or more code units.
- D79 A Unicode encoding form assigns each Unicode scalar value to a unique code unit sequence.
- D94 Unicode encoding scheme: A specified byte serialization for a Unicode encoding form, including the specification of the handling of a byte order mark (BOM), if allowed.
- D16 Higher-level protocol: Any agreement on the interpretation of Unicode characters that extends beyond the scope of this standard

Character names

Consistently refering to Unicode resolves the observation made by NB comment FR-021-01 that the set of character that can be spelled by named escape sequences `\N{}` was, in the standard, a subset of the set of code point with the XID_Continue or XID_Start properties, as the names were derived from ISO and the list of elligible code points from Unicode.

`__STDC_ISO_10646__`

This macro, defined in [cpp] was the most difficult change. Indeed its `yyyymm` is supposed to refer to the publication date of an ISO1646 revision. Here is the thing though. Past discussions in LLVM and the WG14 reflector have established that the wide execution encoding may depend on the C library, and that macro is therefore defined by the C library. On some platforms, the `wchar_t` is actually a runtime property, such that the use of a macro is not suitable.

In addition, its value is not always updated and doesn't indicate much. Indeed, all ISO1646 publications require a 4 bytes `wchar_t` for the storing of code points.

Therefore, we change the wording of that macro to state that its value is implementation-defined, such that we do not have to reference ISO10646 and reword it slightly.

Further work is encouraged to maybe deprecate this macro. Inspecting a large amount of open source code, the only use of the value we found was `(__STDC_ISO_10646__ <= 198700L)`. Of course, there are numerous uses of the macro for its presence.

Note that we don't expect implementers to change the value of that macro, but it lets us not mention ISO 10646 there

Removing the reference to ISO1646:2003

The deprecated `codecvt_utf8` and `codecvt_utf16` refers to UCS-2. Because this has long been obsoleted and removed from ISO 10646, we carry a dependency on ISO 10646:2003.

But, UCS-2 is a strict subset of UTF-16. So we reword the appropriate paragraph to say "code points in the range U+0000-U+FFFF encoded as UTF-16" instead.

Any code point outside that range doesn't satisfy that requirement, and therefore cannot be encoded and causes the conversion to fail with `result::error`, preserving the existing behavior.

Non-Normative reference to UAX #29, Unicode Text Segmentation

Revision 35 (Unicode 12) -> Revision 41 (Unicode 15)

This is used in [format.string.std] for the purpose of width estimation. The only non-editorial applicable change in that document between that version and the last were

- Moved surrogate code points from Control to XX

- Excluded prepended concatenation marks from Control.

Both are bug fixes. Referencing version 15 ensures that the width of recent code points can be correctly estimated. Some implementations refer to Unicode 15 already.

Once we can use the same version, we can remove the reference entierly, as unicode mentions

> This core specification, together with The Unicode code charts, The Unicode Character Database, and The Unicode Standard Annexes, defines The Unicode Standard. The Unicode Standard Annexes form an integral part of The Unicode Standard.

—

Non-Normative reference to UAX #31

Left untouched, someone needs to go through the UAX31 Annex in the C++ standard to ensure consistency. Given this UAX is changing a lot to support source code spoofing mitigation over the next year, we may consider delaying that work? If we do decided to delay that work, we should make sure UAX31 constently refer to the referenced version in the annex.

Wording

🔗 Normative references [intro.refs]

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 2382, *Information technology — Vocabulary*
- ISO 8601:2004, *Data elements and interchange formats — Information interchange — Representation of dates and times*
- ISO/IEC 9899:2018, *Programming languages — C*
- ISO/IEC/IEEE 9945:2009, *Information Technology — Portable Operating System Interface (POSIX)*
- ISO/IEC/IEEE 9945:2009/Cor 1:2013, *Information Technology — Portable Operating System Interface (POSIX), Technical Corrigendum 1*
- ISO/IEC/IEEE 9945:2009/Cor 2:2017, *Information Technology — Portable Operating System Interface (POSIX), Technical Corrigendum 2*
- ISO/IEC 10646, *Information technology — Universal Coded Character Set (UCS)*

- ISO/IEC 10646:2003, *Information technology — Universal Multiple-Octet Coded Character Set (UCS)*
- ISO/IEC/IEEE 60559:2020, *Information technology — Microprocessor Systems — Floating-Point arithmetic*
- ISO 80000-2:2009, *Quantities and units — Part 2: Mathematical signs and symbols to be used in the natural sciences and technology*
- Ecma International, *ECMAScript Language Specification*, Standard Ecma-262, third edition, 1999.
- The Unicode Consortium. The Unicode Standard. Available from: `https://www.unicode.org/versions/latest/`

  [Editor's note: Using recommandation from https://unicode.org/versions/]
- The Unicode Consortium. Unicode Standard Annex, UAX #44, *Unicode Character Database*. Edited by Ken Whistler and Laurenţiu Iancu. Available from: `http://www.unicode.org/reports/tr44/`
- The Unicode Consortium. The Unicode Standard, *Derived Core Properties*. Available from: `https://www.unicode.org/Public/UCD/latest/ucd/DerivedCoreProperties.txt`

  [Editor's note: The reference to The Unicode standard encompasses its annexes and associated]

The library described in ISO/IEC 9899:2018, Clause 7, is hereinafter called the *C standard library*.

The operating system interface described in ISO/IEC 9945:2009 is hereinafter called *POSIX*.

The ECMAScript Language Specification described in Standard Ecma-262 is hereinafter called *ECMA-262*.

[*Note:* References to ISO/IEC 10646:2003 are used only to support deprecated features[depr.locale.stdcvt]. — *end note*]

[...]

🔲 Phases of translation [lex.phases]

The precedence among the syntax rules of translation is specified by the following phases. [*Footnote:* Implementations behave as if these separate phases occur, although in practice different phases can be folded together. — *end note*]

1. An implementation shall support input files that are a sequence of UTF-8 code units (UTF-8 files). It may also support an implementation-defined set of other kinds of input files, and, if so, the kind of an input file is determined in an implementation-defined manner that includes a means of designating input files as UTF-8 files, independent of their content. [*Note:* In other words, recognizing the U+feff byte order mark is not sufficient. — *end note*] If an input file is determined to be a UTF-8 file, then it shall be a well-formed UTF-8 code unit sequence and it is decoded to produce a sequence of ~~UCS~~ Unicode

scalar values ~~that constitutes the sequence of elements of the translation character set~~ . A sequence of translation character set elements is then formed by mapping each Unicode scalar value to the corresponding translation character set element..

For any other kind of input file supported by the implementation, characters are mapped, in an implementation-defined manner, to a sequence of translation character set elements[lex.charset] (introducing new-line characters for end-of-line indicators).

🔲 Character sets [lex.charset]

The *translation character set* consists of the following elements:

- each abstract character ~~named by ISO/IEC 10646, as identified by its unique UCS scalar value~~ assigned a code point in the Unicode codespace, and
- a distinct character for each ~~UCS~~ Unicode scalar value ~~where no named character is assigned~~ not assigned to an abstract character.

[*Note:* ~~ISO/IEC 10646~~ Unicode code points are integers in the range $[0, 10\mathrm{FFFF}]$ (hexadecimal). A surrogate code point is a value in the range $[\mathrm{D800}, \mathrm{DFFF}]$ (hexadecimal). A ~~UCS~~ Unicode scalar value is any code point that is not a surrogate code point. — *end note*]

The *basic character set* is a subset of the translation character set, consisting of 96 characters as specified in [lex.charset.basic]. [*Note:* Unicode short names are given only as a means to identifying the character; the numerical value has no other meaning in this context. — *end note*]

The *universal-character-name* construct provides a way to name other characters.

n-char: one of
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
0 1 2 3 4 5 6 7 8 9
U+002D HYPHEN-MINUS
U+0020 SPACE

n-char-sequence:
n-char
n-char-sequence n-char

named-universal-character:
\N{ *n-char-sequence* }

hex-quad:
hexadecimal-digit hexadecimal-digit hexadecimal-digit hexadecimal-digit

simple-hexadecimal-digit-sequence:
hexadecimal-digit
simple-hexadecimal-digit-sequence hexadecimal-digit

universal-character-name:
\u *hex-quad*
\U *hex-quad hex-quad*
\u{ *simple-hexadecimal-digit-sequence* }
named-universal-character

Table 1: Basic character set

character		glyph
U+0009	CHARACTER TABULATION	
U+000B	LINE TABULATION	
U+000C	FORM FEED	
U+0020	SPACE	
U+000A	LINE FEED	new-line
U+0021	EXCLAMATION MARK	!
U+0022	QUOTATION MARK	"
U+0023	NUMBER SIGN	#
U+0025	PERCENT SIGN	%
U+0026	AMPERSAND	&
U+0027	APOSTROPHE	'
U+0028	LEFT PARENTHESIS	(
U+0029	RIGHT PARENTHESIS	)
U+002A	ASTERISK	*
U+002B	PLUS SIGN	+
U+002C	COMMA	,
U+002D	HYPHEN-MINUS	-
U+002E	FULL STOP	.
U+002F	SOLIDUS	/
U+0030 .. U+0039	DIGIT ZERO .. NINE	0 1 2 3 4 5 6 7 8 9
U+003A	COLON	:
U+003B	SEMICOLON	;
U+003C	LESS-THAN SIGN	<
U+003D	EQUALS SIGN	=
U+003E	GREATER-THAN SIGN	>
U+003F	QUESTION MARK	?
U+0041 .. U+005A	LATIN CAPITAL LETTER A .. Z	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
U+005B	LEFT SQUARE BRACKET	[
U+005C	REVERSE SOLIDUS	\
U+005D	RIGHT SQUARE BRACKET	]
U+005E	CIRCUMFLEX ACCENT	^
U+005F	LOW LINE	_
U+0061 .. U+007A	LATIN SMALL LETTER A .. Z	a b c d e f g h i j k l m n o p q r s t u v w x y z
U+007B	LEFT CURLY BRACKET	{
U+007C	VERTICAL LINE	|
U+007D	RIGHT CURLY BRACKET	}
U+007E	TILDE	~

A *universal-character-name* of the form `\u` *hex-quad*, `\U` *hex-quad hex-quad*, or `\u{`*simple-hexadecimal-digit-sequence*`}` designates the character in the translation character set whose ~~UCS~~ Unicode scalar value is the hexadecimal number represented by the sequence of *hexadecimal-digit* s in the *universal-character-name*. The program is ill-formed if that number is not a ~~UCS~~ Unicode scalar value.

A *universal-character-name* that is a *named-universal-character* designates the character named by its *n-char-sequence*. A character is so named if the *n-char-sequence* is equal to

- the associated character name or associated character name alias or
- the control code alias given in [lex.charset.ucn]. [*Note:* The aliases in [lex.charset.ucn] are provided for control characters which otherwise have no associated character name or character name alias. These names are derived from the Unicode Character Database's `NameAliases.txt`. For historical reasons, control characters are formally unnamed. — *end note*]

[*Note:* None of the associated character names, associated character name aliases, control code aliases have leading or trailing spaces. — *end note*]

A *universal-character-name* that is a *named-universal-character* designates the corresponding character in the Unicode Standard (chapter 4.8 Name) if the *n-char-sequence* is equal to its character name or to one of its character name aliases of type "control", "correction", or "alternate"; otherwise, the program is ill-formed.

[*Note:* These aliases are listed in the Unicode Character Database's `NameAliases.txt`. None of these names or aliases have leading or trailing spaces — *end note*]

[Editor's note: Remove the table "Table 2: Control code aliases [tab:lex.charset.ucn]"]

If a *universal-character-name* outside the *c-char-sequence*, *s-char-sequence*, or *r-char-sequence* of a *character-literal* or *string-literal* (in either case, including within a *user-defined-literal*) corresponds to a control character or to a character in the basic character set, the program is ill-formed. [*Note:* A sequence of characters resembling a *universal-character-name* in an *r-char-sequence* [lex.string] does not form a *universal-character-name*. — *end note*]

The *basic literal character set* consists of all characters of the basic character set, plus the control characters specified in [lex.charset.literal]. [*Note:* The alias BELL for U+0007 shown in ISO 10646 is ambiguous with U+1F514 BELL. — *end note*]

Table 2: Additional control characters in the basic literal character set

character	
U+0000	NULL
U+0007	ALERT
U+0008	BACKSPACE
U+000D	CARRIAGE RETURN

A *code unit* is an integer value of character type[basic.fundamental]. Characters in a *character-literal* other than a multicharacter or non-encodable character literal or in a *string-literal* are encoded as a sequence of one or more code units, as determined by the *encoding-prefix*

[lex.ccon,lex.string]; this is termed the respective *literal encoding*. The *ordinary literal encoding* is the encoding applied to an ordinary character or string literal. The *wide literal encoding* is the encoding applied to a wide character or string literal.

A literal encoding or a locale-specific encoding of one of the execution character sets[character.seq] encodes each element of the basic literal character set as a single code unit with non-negative value, distinct from the code unit for any other such element. [*Note:* A character not in the basic literal character set can be encoded with more than one code unit; the value of such a code unit can be the same as that of a code unit for an element of the basic literal character set. — *end note*] The U+0000 NULL character is encoded as the value `0`. No other element of the translation character set is encoded with a code unit of value `0`. The code unit value of each decimal digit character after the digit `0` (U+0030) shall be one greater than the value of the previous. The ordinary and wide literal encodings are otherwise implementation-defined. For a UTF-8, UTF-16, or UTF-32 literal, the ~~UCS~~ Unicode scalar value corresponding to each character of the translation character set is encoded as specified in ~~ISO/IEC 10646~~ the Unicode Standard for the respective ~~UCS~~ Unicode encoding form.

[...]

🔣 Identifiers [lex.name]

identifier:
identifier-start
identifier identifier-continue

identifier-start:
nondigit
an element of the translation character set ~~of class XID_Start~~ with the Unicode property XID_Start

identifier-continue:
digit
nondigit
an element of the translation character set ~~of class XID_Continue~~ with the Unicode property XID_Continue

nondigit: one of
`a b c d e f g h i j k l m`
`n o p q r s t u v w x y z`
`A B C D E F G H I J K L M`
`N O P Q R S T U V W X Y Z _`

digit: one of
`0 1 2 3 4 5 6 7 8 9`

[Note:The character classes XID_Start and XID_Continue are Derived Core Properties as described by UAX #44 of the Unicode Standard.–end note] The program is ill-formed if an *identifier* does not conform to Normalization Form C as specified in ~~ISO/IEC 10646~~ the Unicode Standard. [*Note:* Identifiers are case-sensitive. — *end note*] [*Note:* In translation phase 4, *identifier* also includes those *preprocessing-token* s[lex.pptoken] differentiated as keywords[lex.key] in the later translation phase 7[lex.token]. — *end note*]

[...]

[lex.string] String literals

[...]

Evaluating a *string-literal* results in a string literal object with static storage duration[basic.stc]. Whether all *string-literal* s are distinct (that is, are stored in nonoverlapping objects) and whether successive evaluations of a *string-literal* yield the same or a different object is unspecified. [*Note:* The effect of attempting to modify a string literal object is undefined. — *end note*]

String literal objects are initialized with the sequence of code unit values corresponding to the *string-literal*'s sequence of *s-char* s (originally from non-raw string literals) and *r-char* s (originally from raw string literals), plus a terminating U+0000 NULL character, in order as follows:

- The sequence of characters denoted by each contiguous sequence of *basic-s-char* s, *r-char* s, *simple-escape-sequence* s[lex.ccon], and *universal-character-name* s[lex.charset] is encoded to a code unit sequence using the *string-literal*'s associated character encoding. If a character lacks representation in the associated character encoding, then the *string-literal* is conditionally-supported and an implementation-defined code unit sequence is encoded. [*Note:* No character lacks representation in any ~~of the UCS~~ Unicode encoding form~~s~~. — *end note*] When encoding a stateful character encoding, implementations should encode the first such sequence beginning with the initial encoding state and encode subsequent sequences beginning with the final encoding state of the prior sequence. [*Note:* The encoded code unit sequence can differ from the sequence of code units that would be obtained by encoding each character independently. — *end note*]
-

[cpp.predefined] Predefined macro names

- `__STDC_VERSION__`
  Whether `__STDC_VERSION__` is predefined and if so, what its value is, are implementation-defined.
- `__STDC_ISO_10646__`
  An integer literal of the form `yyyymmL` (for example, `199712L`). ~~If this symbol is defined, then every character in the Unicode required set, when stored in an object of type `wchar_t`, has the same value as the code point of that character. The *Unicode required set* consists of all the characters that are defined by ISO/IEC 10646, along with all amendments and technical corrigenda as of the specified year and month~~Whether `__STDC_ISO_10646__` is predefined and if so, what its value is, are implementation-defined.

[Editor's note: —– Library wording below —–]

🔲 Formatting escaped characters and strings [format.string.escaped]

A character or string can be formatted as *escaped* to make it more suitable for debugging or for logging.

The escaped string *E* representation of a string *S* is constructed by encoding a sequence of characters as follows. The associated character encoding *CE* for `charT` ([lex.string.literal]) is used to both interpret *S* and construct *E*.

- U+0022 QUOTATION MARK (") is appended to *E*.
- For each code unit sequence *X* in *S* that either encodes a single character, is a shift sequence, or is a sequence of ill-formed code units, processing is in order as follows:
  - If *X* encodes a single character *C*, then:
    - If *C* is one of the characters in [format.escape.sequences], then the two characters shown as the corresponding escape sequence are appended to *E*.
    - Otherwise, if *C* is not U+0020 SPACE and
      - *CE* is ~~a Unicode encoding~~ UTF-8, UTF-16, or UTF-32 and *C* corresponds to either a ~~UCS scalar value whose Unicode property~~ Unicode scalar value whose Unicode property `General_Category` has a value in the groups `Separator` (`Z`) or `Other` (`C`) or to a ~~UCS scalar value whose Unicode property~~ Unicode scalar value with the Unicode property `Grapheme_Extend=Yes`, as described by ~~table 12 of~~ UAX #44 of the Unicode Standard.

        [Editor's note: Merging with P2713R1 [1] should produce the following wording:

        - CE is UTF-8, UTF-16, or UTF-32 and C corresponds to a Unicode scalar value whose Unicode property `General_Category` has a value in the groups `Separator` (`Z`) or `Other` (`C`), as described by UAX #44 of the Unicode Standard, or
        - CE is UTF-8, UTF-16, or UTF-32 and C corresponds to a Unicode scalar value with the Unicode property `Grapheme_Extend=Yes` as described by UAX #44 of the Unicode Standard and C is not immediately preceded in S by a character P appended to E without translation to an escape sequence, or

        ]

        - *CE* is ~~not a Unicode encoding~~ neither UTF-8, UTF-16, nor UTF-32 and *C* is one of an implementation-defined set of separator or non-printable characters

      then the sequence `\u{`*`hex-digit-sequence`*`}` is appended to *E*, where *`hex-digit-sequence`* is the shortest hexadecimal representation of *C* using lower-case hexadecimal digits.
    - Otherwise, *C* is appended to *E*.
  - Otherwise, if *X* is a shift sequence, the effect on *E* and further decoding of *S* is unspecified.

Recommended practice: A shift sequence should be represented in *E* such that the original code unit sequence of *S* can be reconstructed.

- Otherwise (*X* is a sequence of ill-formed code units), each code unit *U* is appended to *E* in order as the sequence `\x{`*hex-digit-sequence*`}`, where *hex-digit-sequence* is the shortest hexadecimal representation of *U* using lower-case hexadecimal digits.

- Finally, U+0022 QUOTATION MARK (") is appended to *E*.

[...]

🔲 Standard format specifiers [format.string.std]

[...]

For the purposes of width computation, a string is assumed to be in a locale-independent, implementation-defined encoding. Implementations should use ~~a Unicode encoding~~ either UTF-8, UTF-16, or UTF-32, on platforms capable of displaying Unicode text in a terminal.

For a string in ~~a Unicode encoding~~ UTF-8, UTF-16, or UTF-32, implementations should estimate the width of a string as the sum of estimated widths of the first code points in its extended grapheme clusters. The extended grapheme clusters of a string are defined by UAX 29. The estimated width of the following code points is 2:

- U+1100 – U+115F
- U+2329 – U+232A
- U+2E80 – U+303E
- U+3040 – U+A4CF
- U+AC00 – U+D7A3
- U+F900 – U+FAFF
- U+FE10 – U+FE19
- U+FE30 – U+FE6F
- U+FF00 – U+FF60
- U+FFE0 – U+FFE6
- U+1F300 – U+1F64F
- U+1F900 – U+1F9FF
- U+20000 – U+2FFFD
- U+30000 – U+3FFFD

The estimated width of other code points is 1.

For a string in ~~a non-Unicode encoding~~ neither UTF-8, UTF-16, nor UTF-32, the width of a string is unspecified.

The *nonnegative-integer* in *precision* is a decimal integer defining the precision or maximum field size. It can only be used with floating-point and string types. For floating-point types this field specifies the formatting precision. For string types, this field provides an upper bound for the estimated width of the prefix of the input string that is copied into the output. For a string in ~~a Unicode encoding~~ UTF-8, UTF-16, or UTF-32, the formatter copies to the output the longest prefix of whole extended grapheme clusters whose estimated width is no greater than the precision.

[...]

[ostream.formatted.print] **Print**

[...]

```
void vprint_unicode(ostream& os, string_view fmt, format_args args);
void vprint_nonunicode(ostream& os, string_view fmt, format_args args);
```

Effects: Behaves as a formatted output function[ostream.formatted.reqmts] of `os`, except that:

- failure to generate output is reported as specified below, and
- any exception thrown by the call to `vformat` is propagated without regard to the value of `os.exceptions()` and without turning on `ios_base::badbit` in the error state of `os`.

After constructing a `sentry` object, the function initializes an automatic variable via

```
string out = vformat(os.getloc(), fmt, args);
```

If the function is `vprint_unicode` and `os` is a stream that refers to a terminal capable of displaying Unicode which is determined in an implementation-defined manner, writes `out` to the terminal using the native Unicode API; if `out` contains invalid code units, the behavior is undefined and implementations are encouraged to diagnose it. Otherwise (if `os` is not such a stream or the function is `vprint_nonunicode`), inserts the character sequence `[out.begin(), out.end())` into `os`. If writing to the terminal or inserting into `os` fails, calls `os.setstate(ios_base::badbit)` (which may throw `ios_base::failure`).

Recommended practice: For `vprint_unicode`, if invoking the native Unicode API requires transcoding, implementations should substitute invalid code units with U+FFFD REPLACEMENT CHARACTER per the Unicode Standard ~~Version 14.0 - Core Specification~~, Chapter 3.9 U+FFFD Substitution in Conversion.

[...]

```
void vprint_unicode(FILE* stream, string_view fmt, format_args args);
```

Preconditions: `stream` is a valid pointer to an output C stream.

Effects: The function initializes an automatic variable via

```
string out = vformat(fmt, args);
```

If `stream` refers to a terminal capable of displaying Unicode, writes `out` to the terminal using the native Unicode API; if `out` contains invalid code units, the behavior is undefined and implementations are encouraged to diagnose it. Otherwise writes `out` to `stream` unchanged. [*Note:* On POSIX and Windows, `stream` referring to a terminal means that, respectively, `isatty(fileno(stream))` and `GetConsoleMode(_get_osfhandle(_fileno(stream)), ...)` return nonzero. — *end note*] [*Note:* On Windows, the native Unicode API is `WriteConsoleW`. — *end note*]

Throws: Any exception thrown by the call to `vformat`[format.err.report]. `system_error` if writing to the terminal or `stream` fails. May throw `bad_alloc`.

Recommended practice: If invoking the native Unicode API requires transcoding, implementations should substitute invalid code units with U+FFFD REPLACEMENT CHARACTER per the Unicode Standard ~~Version 14.0 - Core Specification~~, Chapter 3.9 U+FFFD Substitution in Conversion.

Annex D [depr]

Requirements [depr.locale.stdcvt.req]

For each of the three code conversion facets `codecvt_utf8`, `codecvt_utf16`, and `codecvt_utf8_utf16`:

- `Elem` is the wide-character type, such as `wchar_t`, `char16_t`, or `char32_t`.
- `Maxcode` is the largest wide-character code that the facet will read or write without reporting a conversion error.
- If `(Mode & consume_header)`, the facet shall consume an initial header sequence, if present, when reading a multibyte sequence to determine the endianness of the subsequent multibyte sequence to be read.
- If `(Mode & generate_header)`, the facet shall generate an initial header sequence when writing a multibyte sequence to advertise the endianness of the subsequent multibyte sequence to be written.
- If `(Mode & little_endian)`, the facet shall generate a multibyte sequence in little-endian order, as opposed to the default big-endian order.
- UCS-2 is the same encoding as UTF-16, except that it encodes scalar values in the range U+0000-U+FFFF (Basic Multilingual Plane) only.

  [Editor's note: Surrogates cannot appear in a valid utf-8 sequence (nor in a valid utf-16 sequence)]

For the facet `codecvt_utf8`:

- The facet shall convert between UTF-8 multibyte sequences and UCS-2 or UTF-32 (depending on the size of `Elem`) ~~within the program~~.
- Endianness shall not affect how multibyte sequences are read or written.
- The multibyte sequences may be written as either a text or a binary file.

For the facet `codecvt_utf16`:

- The facet shall convert between UTF-16 multibyte sequences and UCS-2 or UTF-32 (depending on the size of `Elem`) ~~within the program~~.
- Multibyte sequences shall be read or written according to the `Mode` flag, as set out above.
- The multibyte sequences may be written only as a binary file. Attempting to write to a text file produces undefined behavior.

For the facet `codecvt_utf8_utf16`:

- The facet shall convert between UTF-8 multibyte sequences and UTF-16 (one or two 16-bit codes) within the program.
- Endianness shall not affect how multibyte sequences are read or written.
- The multibyte sequences may be written as either a text or a binary file.

~~The encoding forms UTF-8, UTF-16, and UTF-32 are specified in ISO/IEC 10646. The encoding form UCS-2 is specified in ISO/IEC 10646:2003.~~

Bibliography

— ISO 4217:2015, *Codes for the representation of currencies*

— ISO/IEC 10967-1:2012, *Information technology — Language independent arithmetic — Part 1: Integer and floating point arithmetic*

— ISO/IEC TS 18661-3:2015, *Information Technology — Programming languages, their environments, and system software interfaces — Floating-point extensions for C — Part 3: Interchange and extended types*

— The Unicode Consortium. Unicode Standard Annex, UAX #29,
Unicode Text Segmentation [online].
Edited by Mark Davis. Revision 35; issued for Unicode 12.0.0. 2019-02-15 [viewed 2020-02-23].
Available from: `http://www.unicode.org/reports/tr29/tr29-35.html`
[Editor's note: Referencing Unicode encompasses the annexes and the motivation explains that we can and should reference the version of the annex that correspond to The Unicode Standard]

— ~~The Unicode Consortium. Unicode Standard Annex, UAX #31,
Unicode Identifier and Pattern Syntax [online].~~

— The Unicode Standard Version 14.0, *Core Specification*. Unicode Consortium, ISBN 978-1-936213-29-0, copyright ©2021 Unicode, Inc.
Available from: https://www.unicode.org/versions/Unicode14.0.0/UnicodeStandard-14.0.pdf

— IANA Time Zone Database. Available from: https://www.iana.org/time-zones

— Bjarne Stroustrup, *The C++ Programming Language, second edition*, Chapter R. Addison-Wesley Publishing Company, ISBN 0-201-53992-6, copyright ©1991 AT&T

— Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Appendix A. Prentice-Hall, 1978, ISBN 0-13-110163-3, copyright ©1978 AT&T

— P.J. Plauger, *The Draft Standard C++ Library*. Prentice-Hall, ISBN 0-13-117003-1, copyright ©1995 P.J. Plauger

The arithmetic specification described in ISO/IEC 10967-1:2012 is called *LIA-1* in this document.

References

[1] Victor Zverovich. P2713R1: Escaping improvements in std::format. https://wg21.link/p2713r1, 11 2022.

[N4892] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4892